Linköpings Universitet
Institutionen för datavetenskap (IDA)
UPP-gruppen 2023-12-14

Klockslag

Förkunskaper

Du kan skriva imperativa program i C++. Du använder en konsekvent kodstil. Du användar väl valda namn på variabler, funktioner och filer. Dina program använder funktioner. Dina funktioner har ett avgränsat återanvändbart syfte. Du kan föra över data till dina funktioner via parametrar och returvärden. Du använder alltid parameteröverföring istället för globala variabler. Du använder alltid iteration, uppslagning eller funktioner istället för fullständig uppräkning. Du skriver funktioner istället för att upprepa dig. Dina program inkluderar precis de bibliotek som faktiskt används. Du kompilerar alltid dina program med alla kursens varningsflaggor och enligt C++ standard samt åtgärdar all återkoppling från kompilatorn.

Scenario

Du jobbar i ett team på Saab och har fått uppdraget att ta fram en programmodul för ett klockslag. Det är dina kollegor som ska använda din programmodul för att skriva ett flertal olika program. Programmodulen ska skrivas i form av en klass. Du ska representera klockslaget med tre heltal, ett för timme, ett för minut och ett för sekund.

Kvalitetskrav

Du gör allt enligt förkunskaperna. Dessutom uppnår du korrekthet, användbarhet och ändringsbarhet enligt nedan.

Korrekthet

Dina kollegor ska inte kunna använda klockslag på fel sätt. Det ska enbart gå att skapa fullständigt korrekta klockslag. Felaktig användning av din modul ska leda till kompileringsfel i första hand och körfel i andra hand. Din modul ska alltid generera fullständigt korrekt resultat. Din modul ska aldrig generera oväsentligt eller fel resultat. Fel som uppstår när klockslag används ska rapporteras till anropande kod. Anroparen ansvarar för att filtrera vad som rapporteras till slutanvändaren. Du behöver alltså fundera på extra noga på hur din kollega använder din kod och hur hen ska detektera fel som uppstår. Din modul ska ha testfall som alla verifieras av ett helautomatiskt testprogram. Varje del i din klass ska testas utförligt av ditt testprogram.

Användbarhet

Konstanta klockslag ska gå att använda så länge de inte ändras. Programkoden för klockslag ska vara samlad i en implementationsfil med tillhörande headerfil. Det ska vara tydligt vilken kod som är avsedd för dina kollegor att använda och vilken kod de inte ska röra. Användning av ej avsedd kod ska leda till kompileringsfel. Din modul ska följa C++ konventioner där det är möjligt. Din modul ska inte ge upphov till kompileringsvarningar eller kompileringsfel.

Ändringsbarhet

Du ska förutsätta att all din kod som är avsedd att användas av dina kollegor också används i flera olika andra program. Du ska se till att du kan ändra din programmodul så klockslag istället representeras som ett heltal (antal sekunder efter midnatt). En sådan genomgripande ändring av din modul ska inte påverka dina kollegors program över huvud taget. Deras program ska inte märka någon skillnad i syntax eller funktion mellan din gamla och din nya modul. Dina kollegor behöver alltså inte ändra något trots att du ändrat massor.

Funktionalitetskrav

Det ska finnas funktioner i din klass för att:

- Skapa ett klockslag (timme, minut och sekund anges, som tre heltal, eller som sträng) (konstruktor som namnges `Time`)
- Konvertera till sträng formaterad på 24h-format ”hh:mm:ss”. Detta ska vara standardformat om inget annat anges. (namges `to_string`)
- Konvertera till sträng formaterad på 12h-format. Se Tidsformat i tipskapitlet för specifikation. (namges `to_string`)
- Kontrollera om klockslaget är före eller efter lunch (namnges `is_am`)
- Hämta ut aktuell timme, minut eller sekund (namnges `get_hour, get_minute, get_second`)

Det ska finnas operatorer (enligt C++-konvention) i din klass för att:

- Öka och minska med en sekund (prefix och postfix)
- Jämföra med andra klockslag (`< > <= >= == !=`)
- Skriva ut till ström på standardformat (`<<`)

Uppgift

Du ska visa att du förstått hur C++ koncept används och fungerar genom att implementera programmodulen så att kvalitetskrav och funktionalitetskrav uppfylls. Du redovisar muntligen för din assistent under labtid. Därefter lämnar du in digitalt för bedömning. Koncept som ingår listas nedan.

Bonusuppgift: Fler operatorer

Utöka din klass med operatorer (enligt C++-konvention) för att

- Addera med ett helttal N i sekunder (både positivt och negativt) (`+ +=`)
- Subtrahera med ett helttal N i sekunder (både positivt och negativt) (`- -=`)
- Läsa in från ström på standardformat (`>>`)

Du ska undvika duplicering av kod i alla lägen och existerande kod ska från användarens perspektiv fungera oförändrat.

Tillvägagång

1. Läs på och lär dig hur respektive koncept kan användas för att uppnå kraven i scenariot.
2. Planera vilka koncept som behövs till respektive funktionalitet.
3. Planera i vilken ordning funktionaliteten ska implementeras.
   Ordningen är viktig både för TDD och för att kunna nyttja funktioner du redan gjort.
4. Använd testdriven utveckling(TDD) och lös en sak i taget. **Kontrollera tipsen.**
5. Utvärdera om du uppnått varje krav. Nedan finns frågor för självbedömning.
6. Redovisa och lämna in.
7. Din assistent utvärderar om du uppnått kraven. Brister ger komplettering.

Frågor för självbedömning

Om svaret på någon av nedan frågor är "ja" ger det med all sannolikhet komplettering.

- Får du fel eller varningar när du kompilerar med alla kursens flaggor?
- Har du inkluderat en cc-fil någonstans?
- Har du skrivit `using namespace` i din h-fil?
- Har du använt nyckelordet `friend`?
- Har du använt `cout` eller `cerr` någonstans i din klass?
- Får du kompileringsfel om du inkluderar din h-fil två gånger?
- Kan du i huvudprogrammet få ett objekt av din klass att vara ett ogiltigt klockslag?
- Kan du i huvudprogrammet få ett objekt av din klass att vara ett för programmeraren oväntat (men giltigt) klockslag?
- Är någon publik medlemsfunktion svårbegriplig eller irrelevant för den som ska använda klassen?
- Kan du ge argument till `to_string` eller `is_am` som har odefinierad betydelse?

- Skiljer sig parametertyp eller returtyp från C++ standard för någon av dina operatorer?
- Har du någon in-parameter av klasstyp som kopieras eller kan ändras?
- Har du någon medlemsfunktion som aldrig ska ändra sitt objekt, men kan råka göra det utan att det ger kompileringsfel?
- Finns kod som utför jämförelse/strängomvandling på flera ställen?
- Finns någon konstruktor som har en inkomlett datamedlemsinitieringslista?
- Finns någon publik datamedlem?
- Kan du koppla varje listat koncept till något du skrivit i koden?
- Är det olika indentering eller kodstil på olika ställen i koden?
- Saknar du testfall med förväntade indata för någon funktion?
- Saknar du testfall med oväntade indata för någon funktion?
- Saknar du testfall som utvärderar gränsfall för någon funktion?
- Finns det någon rad kod som aldrig körs till följd av dina testfall?

Koncept som ingår

Alla listade koncept (utom frivilliga) ingår i uppgiften. Endast listade koncept ingår i uppgiften.

- Klasser
  - Inkapsling (felsäkert publikt gränssnitt)
  - Skydd (`public`, `private`)
  - Konstruktor (`Time::Time`)
  - Datamedlem
  - Datamedlemsinitieringslista
  - Medlemsfunktion
  - Ändringsskydd på medlemsfunktioner (`const` efter)
  - Skillnad samt likhet mellan klass, instans och objekt
  - Självreferens (`*this`)
- Operatorer
  - Operatoröverlagring (medlem)
  - Operatoröverlagring (fristående, OBS: friend är förbjudet)
  - Utmatning till ström (`operator<<`)
- Övrigt
  - Kasta fånga undantag (`throw`, `std::logic_error`, `CHECK_THROWS()` alt. `try{} catch{}`)
  - Filuppdelning
  - Inkluderingsgard (`#ifndef ...`)
  - Testdriven utveckling(TDD)
  - Catch `https://github.com/philsquared/Catch`
- Imperativ repetition
  - Funktionsöverlagring

  - Standardvärde på parametrar
  - Parameteröverföringsmetoder
    * indata av inbyggd typ: kopia (billigt för inbyggda typer)
    * indata av klasstyp: ej ändringsbar referens (undviker dyra kopior)
    * utdata: ändringsbar referens)
  - Returvärde
  - Referens
- Frivilligt
  - Typkonverterande operator (`operator std::string`)
  - Typkonverterande konstruktor
  - Hindra automatisk typkonvertering (`explicit`)
  - Användardefinierade literaler (`operator""`)

Tips

I detta avsnitt beskrivs testdriven utveckling, hur du hittar testfall och hur du kompilerar effektivt med Catch. Sedan följer exempel på saker som du förväntas testa i laborationen. Vi visar tips på hur valda funktioner fungerar enligt konvention. Exemplen är inte skrivna för just klockslag men principen för hur funktionerna testas är densamma. Dessa tester är inte uttömmande och du ansvarar för att all funktionalitet som efterfrågas i laborationen testas utförligt. Du måste själv fundera ut hur exemplen översätts till klockslag, om det finns fler testfall, och om det finns fler ställen där exemplen kan komma till användning.

Tidsformat

Tidslinjer för de två tidformateringsformaten, 24h tid över med motsvarande 12h tid under. Detta hjälper dig förstå och skapa testfall för tidsformaten:

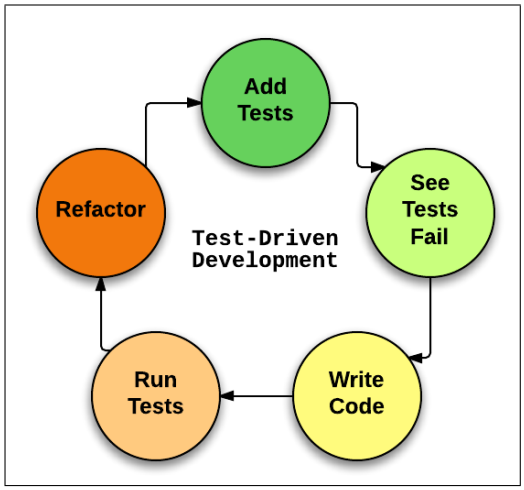
```
24h förmiddag:   00:00:00   ... 00:59:59,   01:00:00   ... 11:59:59
12h förmiddag:   12:00:00am ... 12:59:59am, 01:00:00am ... 11:59:59am

24h eftermiddag: 12:00:00   ... 12:59:59,   13:00:00   ... 23:59:59
12h eftermiddag: 12:00:00pm ... 12:59:59pm, 01:00:00pm ... 11:59:59pm
```

Kursens kompileringsflaggor

```
std=c++17 -Wall -Wextra -pedantic -Weffc++ -Wold-style-cast
```

Arbeta med testdriven utveckling



När du arbetar med denna laboration ska du göra det enligt metoden TDD - testdriven utveckling. I TDD bestämmer du dig för ett litet tillägg till ditt program. Istället för att direkt börja med implementation inleds arbetet med att fundera på hur tillägget ska användas. Därefter skriver du ett program som testar funktionaliteten hos tillägget som om det redan fanns implementerat (du har alltså inte skapat det ännu). Efter att du sett att testet *inte* fungerar implementerar du ditt tillägg steg för steg tills du ser att alla test fungerar. Denna process upprepas för varje liten del av programmet så att testprogrammet blir allt större. För varje nytt tillägg körs alltså alla test, inte bara de du tänkt ut för detta tillägg. Detta för att vara säker på att din ändring inte förstört fungerande kod. Det kan också vara så att du hittar fel i dina tidigare test.
När du skriver ditt testprogram skulle du kunna göra det med vanlig C++-kod som exempelvis if-satser och enkla utskrifter. Det blir dock jobbigt i längden och därför ska du i denna laboration använda dig av ett ramverk som heter Catch. Catch är enkelt att komma igång med och har en bra dokumentation (som anges under koncept som ingår). Du kommer få introduktion till både TDD och Catch under första lektionen.

Komma fram till utförliga testfall

Skriv kod som använder den funktion du just ska implementera. Fundera på vad den funktionen borde ge för resultat med olika indata. Fundera på vilka resultat det inte borde bli med olika indata. Fundera på vilket resultat det skulle kunna bli om det finns en bug. Se sedan till att allt kontrolleras i testfall.
Tänk dig att du vill testa att följande funktion fungerar korrekt:

```
bool is_negative(int i)
{
  return i < 0;
}
```

Då vill du veta att den returnerar ”true” om du skickar in något negativt och ”false” om du skickar in något positivt. Det är lätt hänt att råka göra fel i gränslandet mellan vad som ger resultat ”true” och ”false” så du ska även kontrollera att det blir ”false” om du skickar in 0. Nu har du direkt hittat tre grundläggande testfall. Just “gränslandstester” är särdeles viktiga. Det behövs ofta mer än ett sådant. Så här ser de tre testfallen ut när Catch används:

```
CHECK( is_negative(-1) == true );
CHECK( is_negative(0) == false );
CHECK( is_negative(1) == false );
```

En del i TDD är att faktiskt skriva funktionen som testas fel till en början. Det gör du för kontrollera att testfallen kan upptäcka fel. Det är nämligen lätt att råka skriva även testfallen fel. Genom att medvetet skriva funktionen fel kan felaktiga testfall upptäckas.
Första versionen av funktionen skriver du alltså i TDD som t.ex:

```
bool is_negative(int i)
{
  return true;
}
```

Två av testfallen ska då upptäcka att funktionen inte fungerar korrekt. Därefter implementerar du klart funktionen i trygg förvissning att åtminstone två testfall fungerar. Om alla testfall fungerar för den felaktiga funktionen har du skrivit testfallen fel eller så har du för få testfall.

Kompilering med Catch

Testramverket i `catch.hpp` är stort och du märker snart att det tar lång tid att kompilera. Därför rekommenderas det att du förkompilerar enbart testramverket för att nästa kompilering återanvända den redan kompilerade filen. Du gör detta i två steg; gör först ett huvudprogram som inkluderar testramverket och startar det (givet som `test_main.cc`), och länka sedan ihop det med dina testfall du skriver på en separat fil (t.ex. `time_test.cc`). Här är de kommandon som krävs. Du måste själv lägga till de flaggor som behövs i enligt ovan.

1. Kompilera filen `test_main.cc` men länka inte (skapa inte ett körbart program). Använd kompileringsflagga `-c` för att kompilera utan att länka:
   ```
   g++ -std=c++17 -c test_main.cc
   ```
   Du kommer nu få en fil `test_main.o` om allt gått bra. Detta är en s.k. objektfil med förkompilerad kod redo att länka.
2. Lägg nu till testfall i en separat fil, `time_test.cc`, och din implementation i filerna `Time.h` och `Time.cc`. När du vill kompilera ditt testprogram lägger du till den förkompilerade filen på kommandoraden istället för källkodsfilen så inkluderas den i länkningen. Källkodsfilerna kommer kompileras innan de länkas som vanligt:
   ```
   g++ -std=c++17 test_main.o Time.cc time_test.cc
   ```

Filuppdelning

Det här är kontroller du kan göra själv. Det är inte del i något testprogram. Följande terminalkommandon ska aldrig ge några träffar (det ska inte finnas gch-filer, h-filer ska inte tvinga på användaren namnrymder, cc-filer ska aldrig inkluderas):

```
ls -1l *.gch
grep "using namespace" *.h
grep "#include.*cc" *.cc
```

För att kontrollera din inkluderingsgard kan du helt enkelt prova att inkludera din h-fil två ggr:

```
#include "my_module.h"
#include "my_module.h"
```

Testexempel för klasskonstruktion

Följande konstruktor-deklaration förutsätts finnas för klassen som testas:

```
class my_complex
{
public:
  my_complex(std::string const& val);
};
```

Två förslag på lämpliga tester av konstruktorn:

```
CHECK( my_complex{"0 + i"}.to_string() == "0 + i" ); // korrekt initiering fungerar
CHECK_THROWS( my_complex{"fel"} ); // felaktig initiering ger undantag
```

Testexempel för strängomvandling

Följande deklarationer förutsätts finnas för klassen som testas:

```
enum class notation { cartesian, polar, exponential }; // frivilligt
class my_complex
{
public:
  std::string to_string(bool exp_form) const;
  std::string to_string(notation format) const; // frivilligt
};
```

Tre förslag på hur funktionen kan väntas fungera och testas:

```
my_complex z{0, 16};
CHECK( z.to_string() == "0 + 16i" );
CHECK( z.to_string(true) == "16e^1.5708i" );
CHECK( z.to_string(notation::exponential) == "16e^1.5708i" ); // frivilligt
```

Testexempel för utmatning

Med strängströmmar kan vi skapa en “fejkad” cout och hämta utmatningen som sträng.

```
ostringstream oss{}; // fejkad ``cout'' skapas
my_complex z{"1 + i"};
oss << z << endl;
CHECK( oss.str() == "1 + i" ); // kontroll vad som skrevs ut till ``oss''
```

Notera att vänfunktioner inte är en del av denna laboration. Fundera på om du har något sätt att få fram klockslaget som sträng utan att komma åt datamedlemmarna.

Testexempel för post- och preinkrement

```
int i{0};
int j{i++};
CHECK( i == 1 );
CHECK( j == 0 ); // resultatet av i++ är gamla värdet

int i{0};
int j{++i};
CHECK( i == 1 );
CHECK( j == 1 ); // resultatet av ++i är nya värdet
```

Testexempel för flyttalsjämförelse

Om catch säger att tex 0.42 != 0.42 så har ni troligvis ett litet avrundningsfel på det ena flyttalet. I detta fall måste vi istället kontrollera om det är tillräckligt nära varandra. Detta kan göras genom funktionen nedan.

```
#include <limits>
#include <cmath>

bool compare_equal(double a, double b)
{
    return std::abs(a - b) <= std::numeric_limits<double>::epsilon();
}

TEST_CASE("test case ignoring rounding errors")

{
  double a {0.01};
  dobule b {0.09};

  // CHECK( a + b == 0.1 ); // Avrundningfel!!
  CHECK( compare_equal(a + b, 0.1) );
}
```